An experiment with reactive data-flow tasking in active robot vision

Eric Rutten, Eric Marchand, François Chaumette

An experiment with reactive data-flow tasking in active robot vision

E. P. RUTTEN, E. MARCHAND AND F. CHAUMETTE
IRISA / INRIA - Rennes, F-35042 Rennes, France
(e-mail: {rutten, marchand, chaumette}@irisa.fr*)*

SUMMARY

This paper presents an experiment with the synchronous approach to reactive systems programming, and particularly the SIGNAL language, applied to a significant problem in robot vision: active visual reconstruction. This application consists of the specification of a system dealing with various domains such as robot control, computer vision and transitions between different modes of control. It illustrates the adequacy in such domains of SIGNAL, a data flow programming language and environment. The programming environment features tools for formal specification, analysis, consistency checking and code generation. SIGNAL and its language-level extension for task preemption SIGNAL *GTi* are used at the different levels of the application: data-flow function for the camera motion control (visual servoing), reconstruction method (in parallel to visual servoing, involving the dynamical processes), and reconstruction of complex scenes (with transitions between several robotics tasks). The combination of these levels constitutes a hybrid behavior with (sampled) continuous control and discrete transitions. These techniques are validated experimentally by an implementation on a robotic cell.

KEY WORDS: formal specification language, reactive systems, data flow, task preemption, robotics, active vision.

The synchronous methodology

This paper presents the application in *active robot vision* of the *synchronous approach* to *reactive real time systems* [1], and particularly of the language SIGNAL. Reactive systems are characterized by the fact that their pace is determined by their environment [2]. Their behavior is modelled as a discrete event system, in a way related to the works of Ramadge and Wonham [3]. An interpretation of the synchrony hypothesis is that all the relevant values involved in a computation (input, output and internal) are present simultaneously within the single instant of logical time when the system reacts to its input. In other words, it is valid if the system can be proved to react rapidly enough to perceive all relevant external events. It is an abstraction of the commonly used infinite loop of automatic controllers (input acquisition, computation, output return). This form of synchrony is however a realistic abstraction, since it is actually present for instance in digital hardware (all computations are performed within a clock cycle). It is also current in control theory which is precisely the dedicated application domain of the approach. The control laws are defined in terms of equations on values of the different processed signals at a time t, or, in the case of filters, at times $t-1, t-2, \ldots$: this time index t is shared by different subterms of the equations. It can guarantee deterministic behaviors, in the sense of transition systems: given inputs and a current state, outputs and next state are completely determined. Synchrony facilitates the semantical manipulations on programs, used in the definition of program transformations (e.g., compilation, optimization, distribution) that can be guaranteed to preserve properties of behavior (in particular their determinism). This is an advantage in the context of safety-critical applications, where it is important that the behaviors are *predictable*. This is not the case of real-time operating systems or general purpose languages like Ada [4], because their asynchronous communication mechanisms and tasking constructs are dependent on parameters of the operating system which are unknown or uncontrolled and can not be analysed formally [5].

The synchronous semantics provides for support to a whole set of tools assisting the design of real-time applications, all along their life-cycle. Indeed, the analysis at the different levels of abstraction, from requirements through performance evaluation and optimization, down to code generation (and possibly implementation on specific hardware through co-design): all this is performed on sound formal bases. The analysis and verification techniques handle

the logical time aspects of discrete event systems.

Synchronous languages

A family of languages is based on the synchronous hypothesis, featuring among others ESTEREL [6], LUSTRE [7], and SIGNAL [8]. In STATECHARTS [9], the semantics of parallelism (called orthogonality) is consistent with the synchronous parallel composition. These languages all have complete environments, with sets of tools based upon their formal semantics, and which support specification, formal verification, optimisation and generation of executable code. Their aim is to support the design of safety critical applications, especially those involving signal processing and process control. The synchronous technology and its languages are available commercially, and applied in industrial contexts [10]. Parallelly, research is going on as well as experiments on new features, such as the experiment reported in this paper.

Among synchronous languages, SIGNAL is a real-time synchronized data-flow language [8]. Its model of time is based on instants, and its actions are performed within the instants. SIGNAL *GTi* is an extension that introduces intervals of time, and provides constructs for the specification of hierarchical preemptive tasks executed on these intervals [11]; this way, it offers a multi-paradigm language combining the data flow and tasking paradigms, for hybrid applications blending (sampled) continuous and discrete transition aspects. It is integrated to the environment in the form of a preprocessor, and is compatible with the tools. The SIGNAL programming environment provides users with tools performing the checking of the consistency of synchronizations between data flows (e.g. detection of dependency cycles, i.e. deadlocks), optimisation and automatic code generation. These analyses and transformations are all based on the formal semantics of the language, and provide for an effective and practical formal specification method. From the point of view of debugging, the methodological approach is to perform it at compile time, closer to the original specification, rather than at run-time. Having this guarantee of logical correctness on discrete event behaviors, it is then possible to make very accurate estimations of timing properties of applications, according to the specific hardware architecture which can also be multi-processor; this quantitative analysis is the purpose of the system SYNDEX [12] used in combination with SIGNAL.

Application to active robot vision

This paper presents a real-size experiment of SIGNAL on a robotics system, using this

language extension. It focuses on the programming methodology aspects; a detailed account of the vision methods, made elsewhere [13], is outside the scope of this paper. It is illustrative of the synchronous methodology and its adequateness for that class of systems. It concerns the estimation of the 3-D structure of complex scenes from 2-D images, acquired by a camera mounted on a robot end-effector. The synchrony hypothesis clearly applies to the equations defining a sensor-based control law, and facilitates the implementation of the corresponding control loop. This comes from the fact that the model of time and the equational style of SIGNAL are similar to those of the control theory which forms the framework of the control algorithms. Classical asynchronous languages are less suitable to specify and implement the algorithms involved in this particular vision problem, and in automated control in general. This is because the asynchronous composition of processes represents actual distributed processes and communication channels where there is no notion of global time; however, as was said before, composing systems of equations involves sharing the time index t of signals: hence using an asynchronous language imposes to program the reconstruction of this synchronization. The previous pointsare illustrated here by the use of SIGNAL, at the various levels of the application, for the specification and implementation of the system which deals with various domains such as robot control, computer vision and transitions between different modes of control.

The lowest level concerns the control of the camera motion: the vision system is included in a *control servo loop* as a specific sensor dedicated to a task. At this level, a robot task is seen as a *data flow function* computing the flow of control values for the actuator from the flow of sensor input data. The second level concerns the *optimal estimation of the 3D parameters* describing a geometrical primitive. Embedded in the same formalism, its specification is made in *parallel* with the motion control task, involving also a *dynamical* aspect, in that it is defined in function of past values of observations. The highest level deals with *perception strategies*, where different estimation processes are performed in sequence, depending on conditions. Each structure estimation involves gazing on the considered primitive, hence they have to be performed for each primitive of the scene in sequence, according to a perception strategy. The task-level programming of robots consists in specifying robot tasks and transitions between them by associating them with modes in which they are enabled [14]. SIGNAL *GTi*, the tasking extension of SIGNAL [11], is used for the specification of such nested sequencings of servo-control tasks.

The general motivations for the application of SIGNAL to robot control come from the following observations. A robot control law, at the relatively lowest level, consists in the regulation of a task function, which is an equation $c = f(s)$ giving the value of the control c to be applied to the actuator, in terms of the values s acquired by the sensors. The control of the actuator is a continuous function f, that can be complex. Such a task can be composed of several sub-tasks, with a priority order. The implementation of such a control law is made by sampling sensor information s into a flow of values s_t, which are used to compute the flow of commands c_t: $\forall t, c_t = f(s_t)$. This kind of numerical, data flow computation is the dedicated application domain of data flow languages in general, and of SIGNAL in particular. As indicated by the time index t in this schematical equation, the values involved are simultaneously present, and this is preserved when several such equations are composed. This aspect is adequately handled by the synchrony hypothesis.

THE SIGNAL EQUATIONAL DATA-FLOW REAL-TIME LANGUAGE

SIGNAL [8] is a synchronous real-time language, data flow oriented (*i.e.,* declarative), built around a minimal kernel of operators. This language manipulates signals, which are unbounded series of typed values, and a clock defined as the set of instants when values are present. For instance, a signal `X` denotes the sequence $(\mathtt{x}_t)_{t \in T}$ of data indexed by time t in a time domain T. The clocks of different signals are used to define the qualitative, set-theoretic relation between their presences; for example, for a signal x down-sampled from another signal y, the clock of x is *included in* that of y. Signals of a special kind called `event` are characterized only by their clock *i.e.,* their presence. Given a signal `X`, its clock is noted as `event X`, meaning the event present simultaneously with `X`. The constructs of the language can be used in an equational style to specify the relations between signals *i.e.,* between their values and between their clocks. Systems of equations on signals are built using a composition construct. Data flow applications are activities executed over a set of instants in time: at each instant, input data is acquired from the execution environment. Output values are produced according to the system of equations considered as a network of operations.

The kernel of the SIGNAL language is based on four operations, defining primitive processes, and a composition operation to build more elaborate ones.

- *Functions* are instantaneous transformations on the data. For example, signal Y_t,

defined by the instantaneous function f in: $\forall t,\ Y_t = f(X_{1_t}, X_{2_t}, \ldots, X_{n_t})$ is encoded in SIGNAL by: `Y := f{ X1, X2,..., Xn}`. The signals `Y`, `X1`,..., `Xn` are required to have the same clock.

These functions can be defined within the SIGNAL language (like boolean or arithmetic operations extended to series of values); they can also be defined as *external* functions, to be linked at the compilation of the code generated by the compiler. The latter is a way of calling functions written in C or Fortran for instance, and to use numerical or graphical libraries in connection with a SIGNAL program.

- *Selection* of a signal `X` according to a boolean condition `C` is: `Y := X when C`. The operands and the result do not generally have identical clock. Signal `Y` is present if and only if `X` and `C` are present at the same time and `C` has the value `true`; when `Y` is present, its value is that of `X`.

- *Deterministic merge*: `Z := X default Y` defines the union of two signals of the same type. The clock of `Z` is the union of that of `X` and that of `Y`. The value of `Z` is the value of `X` when it is present, or otherwhise that of `Y` if it is present and `X` is not.

- *Delay*, a "dynamic" process giving access to past values of a signal. For example, equation $ZX_t = X_{t-1}$, with initial value V_0 defines a dynamic process which is encoded in SIGNAL by: `ZX := X$1` with initialization `ZX init V0`. Signals `X` and `ZX` have the same clock. Derived operators include delay on N instants (`$N`), and a `window M` operation giving access to a whole window of past values (from instants $t - M$ to t), as well as combinations of both operators.

- *Composition* of processes is the associative and commutative operator "`|`" denoting the union of the underlying systems of equations. In SIGNAL, for processes P_1 and P_2, it is written: `(|` P_1 `|` P_2 `|)`. The semantics of primitive processes is given by the solution of the system of equations, in terms of sets of the possible traces for the involved signals. Synchronous composition corresponds exactly to the composition of systems of equations, where the resulting semantics is the solution of the resulting equation system, i.e. the traces which are solution to both the composed equations systems, in other terms the intersection of the solutions to each of them. It can be interpreted as parallelism, with signals supporting instantaneous communication (sharing the same time index t) between processes.

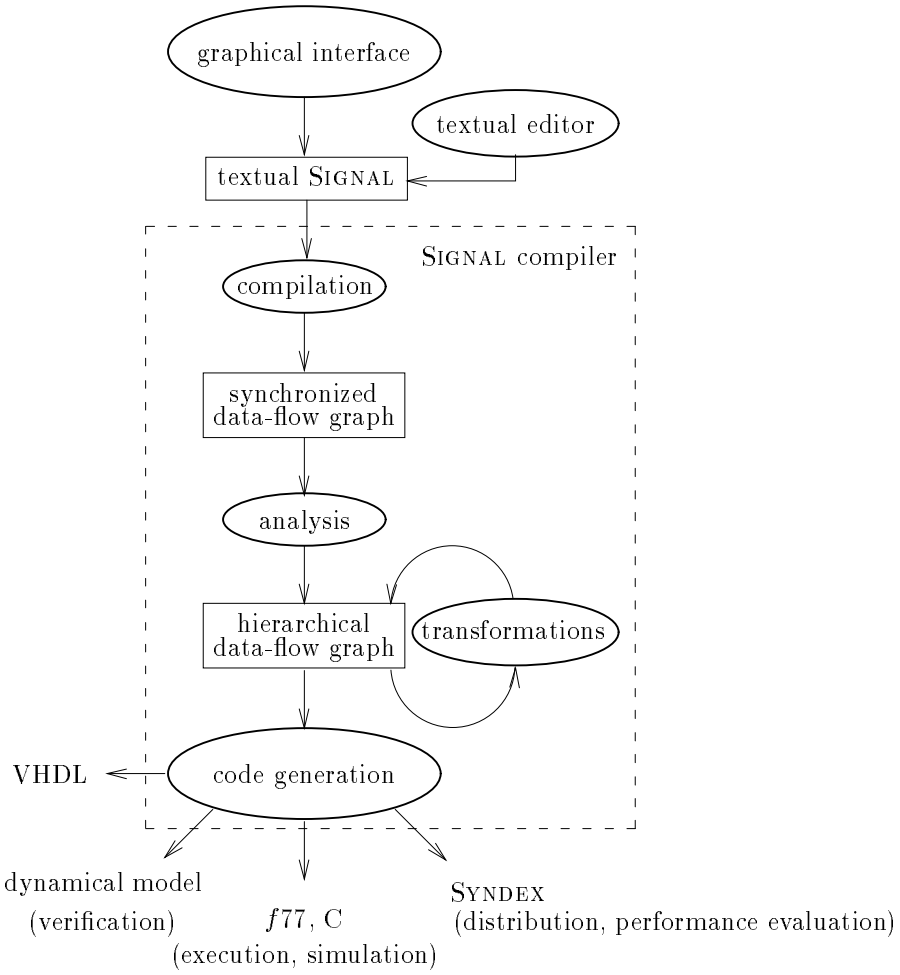


Figure 1: The SIGNAL design environment.

For example, a filter defined by equation $y_t = (x_t + x_{t-1} + x_{t-2})/3$ which can also be written $y_t = (x_t + zx_t + zzx_t)/3, zx_t = x_{t-1}, zzx_t = x_{t-2}$, is specified in SIGNAL by: `(| Y := (X + ZX + ZZX)/3 | ZX := X$1 | ZZX := X$2 |)`. As in the equational definition, and in contrast to imperative languages, the order in which the equations are given is immaterial: it does not change the values denoted.

Derived processes have been defined from the primitive operators, providing programming comfort. For instance, `synchro{X,Y}` specifies the synchronization of signals `X` and `Y`; `when C` gives the clock of occurrences of `C` at the value `true`; `X cell B` memorizes values of `X` and outputs them also when `B` is true; the expression `X := #Unit init V0` is a counter of the occurrences of `Unit`. Arrays of signals and of processes have been introduced as well. Hierarchy, modularity and re-use of processes are supported by the possibility of defining process models, and invoking instances.

The SIGNAL design environment is a set of tools around the SIGNAL compiler, which is itself more than a translator to object code, as shown in Figure 1. The SIGNAL compiler performs the analysis of the consistency of the system of equations, and determines whether the synchronization constraints between the clocks of signals are verified or not. This is based on an internal representation featuring a graph of data dependencies between operations, augmented with temporal information coming from the clock calculus: it is called the synchronized data-flow graph in Figure 1. This formal symbolic analysis on the specifications supports the detection of non-deterministic behaviors, cycles of dependencies between signals and logical incoherences. Transformations are applied to the graph in order to build a hierarchy of the clocks of the program following their inclusion relations. Optimizations feature the detection of null clocks, and removal of actions associated to that clock, which would be dead code. If the program is constrained so as to compute a deterministic solution, then executable code can be automatically produced (in C or FORTRAN). Other output languages are SYNDEX (an environment devoted to the distribution and performance evaluation of data flow algorithms [12]), VHDL (which can be connected to hardware design environments), and a formal model of the dynamical behavior of SIGNAL programs, connected to a *proof system*, to verify dynamic properties of programs, involving state information (in the delayed signals) and transitions in reaction to occurrences of other signals. This way, it is possible to formally specify and verify the satisfaction of dynamical properties of the behaviors; this has been applied to a production cell controller [15]. The complete programming environment also contains a graphical, block-diagram oriented user interface where processes are boxes linked by wires representing signals, as illustrated in Figure 2(a).

The Control and Estimation Algorithms

General issues

This section gives a very simplified presentation, corresponding to the focus of this paper, of the type of computations involved in the vision-based algorithms, for which general presentations can be found in [16, 17, 18]. It concentrates on the aspects that are important from the point of view of programming. The types of data handled are vectors and matrices of reals, and the operations performed are arithmetic, inversion, etc ... The set of operations is to be performed on each input data *i.e.*, at each reaction instant in the logical time. It corresponds to the control theory equations (given in continous time) adapted to the discretization of sampled sensor values. The considered algorithms have two specific

features:

- First, they have an equational nature: they express relations between various flows of data, in a declarative way. In particular, the iterative aspect in the control loop (at each instant) is completely implicit.
- Second, they are synchronous: the equations involve values of the different quantities at the same logical instant.

Classical programming methods are not so well adapted to specifying and programming such algorithms. Asynchronous imperative languages require the explicit management of low level aspects of the implementation (like the sequencing of computations imposed by data dependencies), and of the temporal aspects (e.g., down-samplings on a flow of data, multi-rate parallel computations), for which there is no well-founded support or model. In the case of this application, the composition is not needed for the programming of implementation parallelism, but of specification parallelism. Therefore, using asynchronous communication would introduce unnecessary complexity in the programming. What SIGNAL offers here is an adequate high level of abstraction for declarative specification, as well as a coherent and powerful model of time.

Visual Servoing Control

The control of the camera is performed in a *control servo loop* where a vision system is included as a specific sensor dedicated to a task. In order to present the kind of equations that are programmed, we describe a general and simple control law for the positioning with respect to a static object [17], which illustrates the kind of equations that are programmed:

$$T_c \;=\; -\lambda A_1 C(S - S_d) \;-\; \lambda A_2 e_2 \;-\; A_2 \frac{\partial e_2}{\partial t} \qquad (1)$$

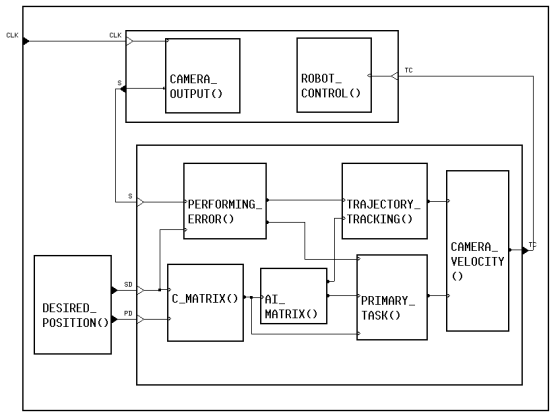
where:

- T_c is the velocity to be given to the camera according to the control law
- S describes the current position of the object in the image
- S_d represents the desired value to be reached by S
- C is a matrix which depends on the position of the objects in the image as well as its shape and 3D position, expressed here by the corresponding parameters S_d and p_D.
- e_2 is a *secondary task* such as a trajectory tracking

- A_1 and A_2 are two projection operators which depend on C and ensure the realization of the secondary task under the constraint that the primary task $(S - S_d)$ is achieved.
- λ is a gain to be tuned.

In other words, the control law consists of the computation of a velocity, depending on the one hand on a *primary task* which aims at lowering the *error* $(S - S_d)$, and on the other hand on a *secondary task* e_2 (in our case a trajectory tracking); both sub-tasks interact in such a way that the secondary task is performed only when it does not go against the primary task.

Figure 2(a) shows the modular description, in SIGNAL (using the graphical interface of the programming environment), of a general visual servoing process and the corresponding SIGNAL program in its textual form is depicted in Fig. 2(b).



(a) SIGNAL graphical specification.

```
(|(| S := CAMERA_OUTPUT{CLK}
   | ROBOT_CONTROL{Tc}
   |)
 |(| {error,acc} :=
           PERFORMING_ERROR{S,SD}
   | C := C_MATRIX{PD,SD}
   | {A1,A2} := AI_MATRIX {C}
   | tau := PRIMARY_TASK{C,A1,error}
   | traj := TRAJECTORY_TRACKING{A2,acc}
   | Tc := CAMERA_VELOCITY{tau,traj}
   |)
 | {PD,SD} := DESIRED_POSITION{}
 |)
```

(b) Equational specification in SIGNAL.

Figure 2: Modular description of a general visual servoing process.

At a high level of the modular description, the visual servoing process is composed of three different sub-modules. A `CAMERA_OUTPUT` module produces a flow `S` of image data at video rate given by the signal `CLK`. The signal `S` has the clock `CLK`; this data, as well as the desired 2D position `SD` and 3D position `PD` delivered by the `DESIRED_POSITION` module, are received as input by the control module. In the subprocesses of the control module, they are involved in computations with `S`, hence `SD` and `PD` will be given the same clock `CLK` by the clock calculus in the compiler. The control module process computes the corresponding camera velocity `Tc`. This camera velocity is transmitted to the `ROBOT_CONTROL` module; it consists basically of a call to a function writing the command in the device's command register, therefore it has no output visible at this level of specification. For the same reason,

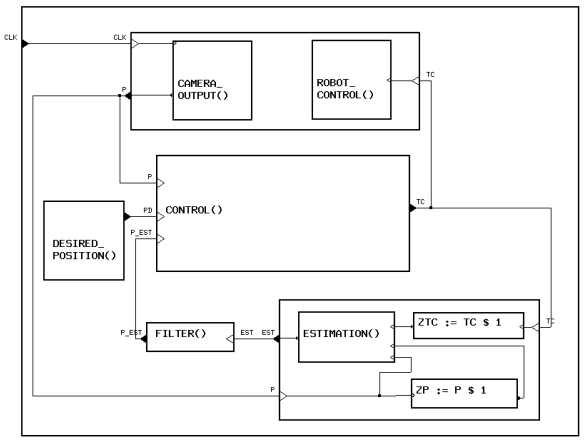
the camera module has no visible image input, because the image from which S is computed is acquired by a function call to the image buffer.

The control module itself is hierarchically decomposed into sub-modules. The process named PERFORMING_ERROR computes two signals: error is the difference between the desired position given by signal SD and the sensed value S, and acc is an event present when a given accuracy is reached, i.e. when the error is less than a given threshold. The C_MATRIX process computes the matrix C. From the output of this module, a process named AI_MATRIX computes the two matrices A_1 and A_2. C and A1 are used in combination with the output error of the PERFORMING_ERROR module to determine a component tau of the camera velocity in the process PRIMARY_TASK; using A2 and acc, the module TRAJECTORY_TRACKING module performs the secondary task computing another component traj. This trajectory tracking is performed only when the event acc is present. The final velocity Tc is then computed by process CAMERA_VELOCITY using the two flows of data coming from the PRIMARY_TASK and the secondary TRAJECTORY_TRACKING task.

These sub-modules are processes themselves, instanciations of process models. Differently from function calls, their inputs and outputs are not necessary all synchronized at the same clock. For example, the signal acc is only present when the value of error reaches a given accuracy, i.e. it is present at a clock *included in* that of the error; hence it is also included in that of A2, which is computed at all instants. The control module itself can be declared as the body of a process model, with three inputs and one output, named CONTROL, and can then be re-used by instanciation, as is illustrated next.

Estimation of Structure From Controlled Motion

The recovery of the 3-D description of a scene from a sequence of images is one of the main issues in computer vision. One approach, called *dynamic vision*, consists in using the measure of the camera motion for the 3-D structure estimation of objects featured in a *sequence* of images. In order to obtain a better accuracy, the camera motion has to be controlled: this is then called *active vision*. It can involve fixing at and gazing on the object (i.e. it must have a constant and particular position in the image), adding more constraints on the control of the camera motion [18]. The estimation method is based on the use of the current and the past values of the position of the object in the image (*i.e* P_t and P_{t-1}). Furthermore, the value of the camera velocity between these two instants t and $t-1$ must be measured. In SIGNAL, the past value of P and the camera velocity can be expressed using the delay operator $. In addition, the output of this process is smoothed, by computing



(a) SIGNAL graphical specification.

```
(|(| P := CAMERA_OUTPUT{CLK}
   | ROBOT_CONTROL{Tc}
   |)
 | Tc := CONTROL{P,Pd,p_est}
 | Pd := DESIRED_POSITION{}
 | p_est := FILTER{est}
 |(| ZTc := Tc$1
   | ZP := P$1
   | est := ESTIMATION{P,ZP,ZTc}
   |)
 |)
```

(b) Equational specification in SIGNAL.

Figure 3: Control and estimation in parallel.

the mean value of the current estimation and of the two previous ones. The control process `CONTROL` presented previously is reused. The estimation process is added to it in such a way that it is executed in parallel with the control law, as shown in Fig. 3. Textually, the program is shown in Fig. 3(b).

This is an example of the significance of the synchronous hypothesis in the framework of such applications. Indeed, in order to improve the behavior of the control law, the C matrix is here computed using each new value provided by the measurement and the estimation processes (`P` and `p_est` instead of `Pd` and `pd`). According to the synchrony hypothesis, the value at instant t of the C matrix is updated using the estimated 3D parameters and the current position of the primitive in the image at the *same logical instant* t.

SIGNAL*GTi* TASKS FOR THE RECONSTRUCTION STRATEGY

The previous section gave a framework for the specification and implementation in SIGNAL of vision-based tasks as well as estimation algorithms. Once a library of such modules is available, the specification of higher-level, more complex behaviors requires the possibility to combine these tasks in various ways. Especially, one wants to combine them in sequences, starting and interrupting them on the occurrence of events, that can be either external (coming from logical sensors) or internal (e.g., reaching certain thresholds). This level of robot programming necessitates preemption structures for concurrent tasks. The purpose of SIGNAL*GTi* is precisely to augment SIGNAL with objects and operations for the construction

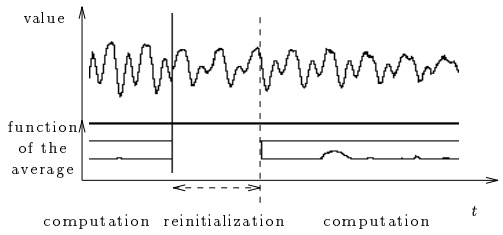


Figure 4: Phases in speech signal processing.

of such preemptive hierarchies of data flow tasks. These extensions are defined on top of SIGNAL, and handled by a pre-processor. In that sense, being translated into SIGNAL, they are compatible with the tools of the rest of the environment.

THE PREEMPTION OF DATA-FLOW TASKS IN SIGNAL*GTi*

SIGNAL*GTi* is an extension to SIGNAL, handling tasks executing on time intervals and their sequencing [11]. The motivation is to provide ways of representing behaviors switching between *different modes of continuous interaction* with their environment. These modes are identified by time intervals delimited by discrete start and end events, and within which tasks are executed. The application domain is the control of physical processes e.g. signal processing or robotics, featuring both computations on flows of sensor data, and discrete transitions in a control automaton.

For example in a speech recognition system [8], the processing of the acoustic signal features a segmentation treatment: boundaries of the segments are determined by changes in the signal. These are detected by comparison with an average value, which is computed on a time window on its past values. Such an application presents successive modes or phases: a phase of initialization must compute the value of this average, and then the regular computation can be performed. Hence two phases (reinitialization and computation) alternate on complementary, periodic time intervals as shown in Figure 4. The sequencing between these phases is intrinsic: it is imposed by dependencies on results produced and consumed. Our goal is to provide a programmer with language constructs enabling the explicit designation of the phases in a process. This is achieved by subdividing its activity interval into sub-intervals for the different modes, and associating sub-activities to each of them.

In SIGNAL*GTi*, data flow and sequencing aspects are both encompassed in the same

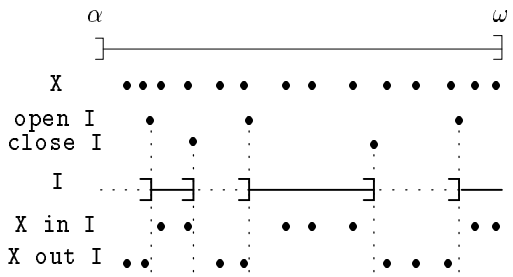
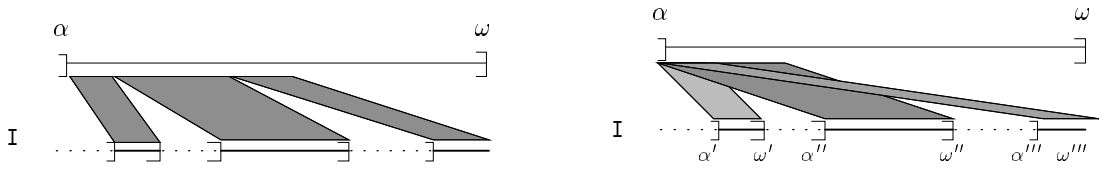


Figure 5: Time intervals sub-dividing]α,ω].

language framework, thus relying on the same model for their execution and analysis (for the compilation and verification of correctness of programs). In this approach, a data flow application is considered to be executed from an initial state of its memory at an initial instant α; it is before the first event of the reactive execution. A data flow process has no termination specified in itself: therefore its end at instant ω can only be decided in reaction to external events or the reaching of given values. Hence ω is part of the execution, and the time interval on which the application executes is the left-open, right-closed interval]α,ω].

Time intervals are introduced in order to enable the structured decomposition of]α,ω] into left-closed, right-open intervals as illustrated in Figure 5, and their association with processes [11]. An interval `I` is delimited by occurrences of bounding events at the beginning `B` and at the end `E`. It has the value `inside` between the next occurrence of `B` and the next occurrence of `E`, and `outside` otherwise. It has an initial value `I0` (`inside` or `outside`). This is written: `I := ]B, E] init I0`. Like]α,ω], sub-intervals are left-open and right-closed. This choice is coherent with the behavior expected from reactive automata: a transition is made according to a received event occurrence and a current state, which results in a new state. Hence, the instant where the event occurs belongs to the time interval of the current state, not to that of the new state. The operator `compl I` defines the complement of an interval `I`, which is `inside` when `I` is `outside` and reciprocally. Operators `open I` and `close I` respectively give the opening and closing occurrences of the bounding events. Occurrences of a signal `X` inside interval `I` can be selected by `X in I`, and reciprocally outside by `X out I`. In this framework, `open I` is `B out I`, and `close I` is `E in I`.

Tasks consist in associating some (sub)process of the application with some (sub)interval of]α,ω] on which it is executed. Traditional processes in SIGNAL are tasks active on]α,ω]: they are *persistent* throughout the whole application. Inside the task interval, the task process is active *i.e.*, present and executing normally. Outside the interval, the process is inexistent *i.e.*, absent and the values it keeps in its internal state are unavailable. In some



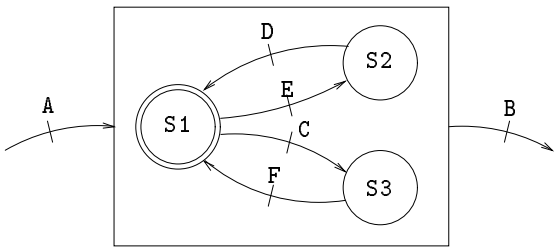
(a) Task `on` interval `I`.

(b) Task `each` interval `I`.

Figure 6: Tasks associating a time interval with a process.

sense, it is out of time, its clock being cut. Tasks are defined by the process `P` to be executed, the execution interval `I`, and the starting state (current, or initial) when (re-)entering the interval. More precisely, the latter means that, when re-entering the task interval, the process can be re-started from its current state at the instant where the task was *suspended* (*i.e.,* in a temporary fashion): this is written `P on I`. Figure 6(a) illustrates that the possible behaviors of task `P on I` are those that process `P` would have had on interval $]\alpha,\omega]$, but they are split in time on the successive occurrences of interval `I`. Alternately, a task can be re-started from its initial state as defined by the declaration of all its state variables, if the task was *interrupted* (meaning: aborted in a definitive fashion): `P each I`. In this case, as illustrated in Figure 6(b), the behaviors of task `P each I` are like prefixes of those that `P` would have had on $]\alpha,\omega]$, on each of the successive occurrences of interval `I`. In that sense, each of these successive occurrences of `I` is a new $]\alpha',\omega']$, $]\alpha'',\omega'']$, ..., for `P`. The processes associated with intervals can themselves be decomposed into sub-tasks: this way, the specification of *hierarchies* of complex behaviors is possible.

Task sequencing and preempting is achieved as a result of constraining intervals and their bounding events, and associating activities to them by constructing hierarchical tasks. *Parallelism* between several tasks is obtained naturally when tasks share the same interval, or overlapping intervals. *Sequencing tasks* then amounts to constraining the intervals of the tasks, by constraining their bounding events. Using `on` and `each`, as defined above, enables control of activities and more elaborate behaviors can be specified. This way, it is possible to specify hierarchical parallel place/transition systems. Each time interval holds some state information, and events cause transitions between these states. For example, in the behavior illustrated in Figure 7(a), a transition leads from the initial place `S1` to place `S2` on the occurrence of an event `E`, except if the event `C` occurs before, leading to place `S3`. If `E` and `C` happen synchronously or are constrained to be equal, then both places `S2` and `S3` are



(a) Hierarchical place/transition system.

```
(| S1 := ](D in S2 default F in S3),
           E default C] init inside
 | S2 := ]E in S1, D] init outside
 | S3 := ]C in S1, F] init outside
 |) each ]A, B]
```

(b) Specification in SIGNAL*GTi*.

Figure 7: Task sequencing and preempting in SIGNAL*GTi*.

entered. This is a sub-behavior attached to a place entered upon event `A` and left upon event `B`. This can be coded by a task and intervals such that the closing of the one is the opening of the other, as in the code shown in Figure 7(b). This example illustrates a hierarchy of tasks and intervals; it could also have featured data-flow equations. This is the advantage of embedding such constructs into a data-flow language and environment: it enables the integration of the two aspects for the specification of hybrid applications.

The encoding of time intervals and tasks into the SIGNAL kernel [11] is implemented as a pre-processor to the SIGNAL compiler, called SIGNAL*GTi*. It has also been used in the specification and implementation of a model of the controller of a power transformer station and behavioral animation in a computer graphics-based simulation environment.

Reconstruction Strategies for Complex Scenes

Sequencing vision tasks

The purpose of the vision application under consideration is the reconstruction of environments composed of several objects such as cylinders and polyhedral objects. As mentioned in the previous section, the camera fixates at and gazes on it and performs a particular motion using active vision, in order to obtain a precise and robust estimation of the structure of a selected primitive,. So, the estimation has to be successively performed for each primitive of the scene, with different phases: selection of a primitive, precise active estimation, and concurrently, coarse estimation of the other ones, as well as the creation or the update of a list of 2-D segments which contains a 2-D description of the observed scene. For this, *tasks* are defined that associate the structure estimation and other processes with a *time interval* on which they are active (see Figure 8). The transitions between tasks are discrete events and are function of the image data, the estimated parameters of the primitives, and the

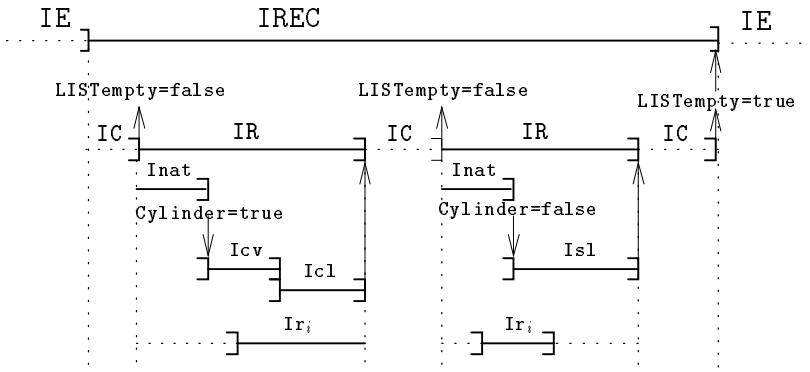


Figure 8: Specification of the sequencing in terms of activity intervals: a possible trace.

state of the list of segments. Concerning termination of estimation processes, each active estimation ends when all the primitive parameters have been accurately computed with a sufficient precision. Each coarse estimation ends when the corresponding segment gets out of the image or when the active estimation ends. After each estimation, the list of 2-D segments is updated, as well as the 3D map of the scene, a new segment is chosen and an other estimation is performed.

Figure 8 illustrates the specification of the behavior of the system using a possible execution trace. An *exploration* phase computes a new camera position and detects in the corresponding image the list of 2-D segments; it is active during the time interval `IE`. In alternance with this, *i.e.*when not in `IE`, the *scene reconstruction* process is active on `IREC`. It is itself an alternance between a *primitive selection* process, on interval `IC`, and the *primitive reconstruction*. The selection chooses a segment in the list to be considered. If the list is empty (*i.e.*, `LISTempty=true`), it causes the *scene reconstruction* (`IREC`) to exit. When the list is not empty, the primitive reconstruction process for the chosen primitive on `IR` is itself decomposed into sub-activities. It begins with a recognition process which estimates the *nature* of the considered primitive (segment or cylinder), on `Inat`, which ends with the boolean event `Cylinder`. It continues with the estimation of the parameters of its 3D structure (according to the value of `Cylinder`: in the case of a segment (`Cylinder=false`), only its length on interval `Isl`; in the case of a cylinder (`Cylinder=true`), its radius and position of axis on `Icv`, and then its length on `Icl`). In parallel with this estimation, a coarse estimation of some primitives can be performed on intervals `Ir`$_i$. After each estimation of a primitive, the list of 2D segments is updated and a new selection is performed on `IC`. Each vision-based task incorporated in this scheme is a data-flow task based on the visual servoing approach. They are implemented as described previously.

Termination and parallelism

Interesting points of the specification in SIGNAL *GTi* are the treatment of the termination and sequencing of vision data-flow tasks, and that of the parallelism between them.

A data-flow process defines, like our vision tasks, a behavior, but not a termination: this aspect must be defined separately. One way of deciding on termination of a task is to apply criteria for reaching a goal depending on a condition involving acquired sensor values or computations (e.g. a given precision is reached). The evaluation of this condition must be performed at all instants: hence this evaluation is another data flow treatment. The instant when the condition is satisfied can be marked by a discrete event, which, causing termination of the task, can also cause a transition to another task at the higher level of the reactive sequencing. In this sense, this event can be used to specify the end of the execution interval of the task. Evaluation of such conditions can be made following a dynamic evolution: a sequence of modes for evaluation of the condition can be defined, becoming finer (and possibly more complex) when nearing interesting or important values.

Parallelism between two tasks is transparent to the programmer using the composition operator. This is the case, for example, of the coarse estimation process and the active estimation process. To perform these estimations, they both use the same information (*e.g.* the measure of camera velocity, the image data at current and previous instants), in such a way, according to the synchrony hypothesis, that they can use it at the same logical instant. In fact, it is a parallelism of specification, and the compiler manages all the synchronization and communication between tasks.

Part of the specification in SIGNAL *GTi* corresponding to Fig. 8 is given in Fig. 9, in a simplified form keeping only the essential aspects, for the sake of brevity and readability (i.e. skipping declarations and some of the structure of the actual program). It does not detail the processes associated to the intervals, which can be described as follows:

- `Exploration`, which builds a list of 2-D segments, and outputs the boolean signal `LISTempty` at the value `false`, hence ending interval `IE`.
- `Choice` outputs the boolean signal `LISTempty`: the occurrence of this signal ends interval `IC`.
- `Nature` outputs the boolean signal `Cylinder`, which ends interval `Inat`. The value of `Cylinder` is `true` if the primitive is a cylinder, `false` if it is a segment.
- `Cylinder_vertex`, `Cylinder_length` and `Segment_length` output the respective pre-

```
Application:
(| IE := ] when LISTempty, when not LISTempty] init inside
 | IREC := comp IE
 | Exploration each IE
 | Structure_estimation each IREC
 |)
```

```
Structure_estimation:
(| IC := ] close Icl default close Isl, LISTempty] init inside
 | IR := comp IC
 | Choice each IC
 | Primitive_estimation each IR
 |)
```

```
Primitive_estimation:
(| Optimal_estimation
 | (| Coarse_estimation_1 | ... | Coarse_estimation_n |)
 |)
```

```
Optimal estimation:
(| Inat := ] LISTempty, Cylinder] init inside
 | Nature each Inat
 | Icv := ] when Cylinder, when (|prec_cv|< ε_cv)] init outside
 | Cylinder_vertex each Icv
 | Icl := ] close Icv, when (|prec_cl|< ε_cl)] init outside
 | Cylinder_length each Icl
 | Isl := ] when not Cylinder, when (|prec_sl|< ε_sl)] init outside
 | Segment_length each Isl
 |)
```

```
Coarse_estimation_i:
(| Ir_i := ] New_Segment, Segment_Lost ] init outside
 | Coarse_estimation each Ir_i
 |)
```

Figure 9: Specification of the reconstruction strategy.

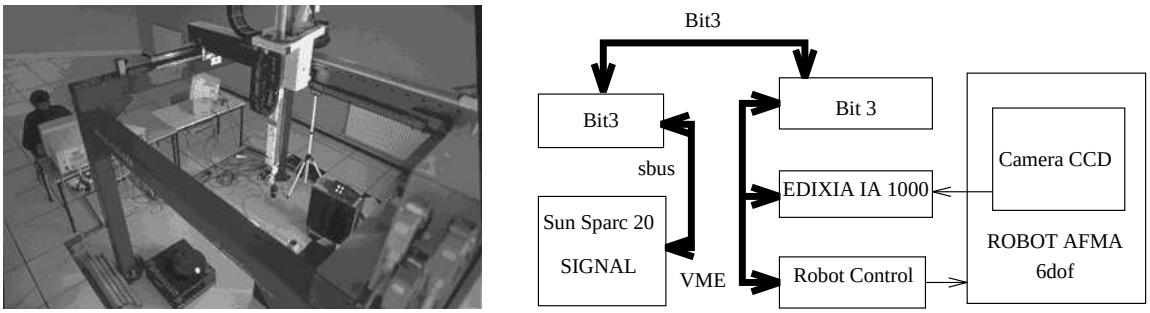
cision measures $prec_i$ which, when they reach a desired value ε_i, end the respective intervals.

- The tasks `Coarse_estimation`$_i$ perform a sub-optimal estimation of segments in the image other than the chosen one. An instance i of it is started on the event of the detection of a segment: `New_Segment`. It is stopped when the segment disappears from the image (event `Segment_Lost`). Several instances can be active in parallel. The active coarse estimation tasks are all preempted at the end of the optimal estimation task i.e., when leaving `IR`.

IMPLEMENTATION IN A ROBOTICS CELL

The whole application presented in this paper has been implemented on an experimental testbed composed of a CCD camera mounted on the end effector of a six degrees of freedom cartesian robot (see Fig. 10(a)). The image processing part is implemented in C and performed on a commercial image processing board (EDIXIA IA 1000). The implementation of

the control law, the 3D structure estimation and the sequencing was carried out using the SIGNAL language running on a SPARC 20 workstation. Fig. 10(b) shows our testbed robot architecture.



(a) Camera mounted on a 6 dof robot.

(b) Architecture.

Figure 10: The experimental robotic cell.

Figure 11 shows a graphical view of the reconstruction environment, which was built using OSF/Motif. The event-based management of this graphical interface is also programmed in SIGNAL; only X11 functions are defined and called as external processes. In Fig. 11, looking from the bottom to the top of the environment, the following are represented: the current state of the different time intervals (*i.e.*, activity of tasks), the evolution of the parameters describing the selected primitive, the error between the current and the desired position of the primitive in the image, an automaton-like representation of the behavior, the image with the list of segments superimposed on it and finally the current representation of the reconstructed scene.

RELATED WORK AND DISCUSSION

RELATED WORK

A review of the techniques classically used for real-time programming is given in [5]. The application of real-time techniques to vision applications is reviewed in [19]. The most common model of time for concurrent programming is asynchrony (see e.g., [4, 20, 21]). Two main approaches raised: one is based on formal methods (such as finite state automata or Petri Net), the other is based on tools able to express concurrency (real time OS, or concurent programming languages). The finite state automata are well known tools, deterministic, efficient and they allow the formal verification of properties. However, the composition of little automata can yeld a very big one often impossible to understand; furthermore a little

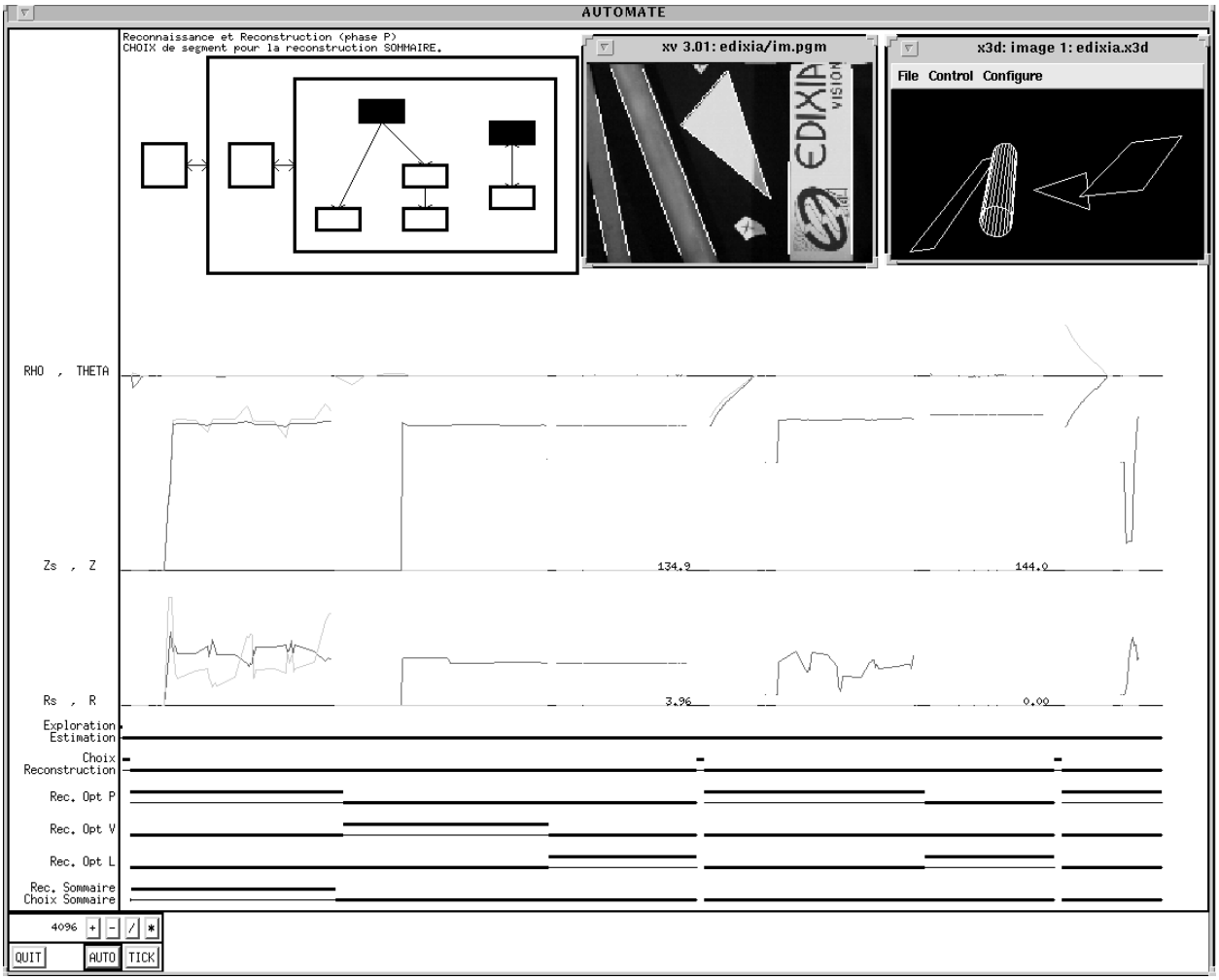


Figure 11: The synchronous environment for 3D scene reconstruction.

change in the specification can provoke a deep transformation of the automaton. Finally, it can be pointed out that the expression of parallelism and preemption of tasks are not supported by this formalism. Petri Nets are often used for small applications and if they support concurrency, they do not support hierachical design, and they are not deterministic.

The second approach is based on the expression of concurrency. Concurrent programming languages such as OCCAM (inspired by asynchronous composition in CSP) or ADA have numerous advantadges. They are well structured and allow a good modularity. But, unlike for synchronous languages [5], the synchronisation between processes is made according to a composition which takes into account the asynchrony of communications as performed during execution. This makes that the transitions made by composed processes can be made at different rates, and the resulting state is unpredictable; hence the behavior is non-deterministic. This is a problem for their use in the implementation of reactive systems, notably if they are to be deterministic w.r.t. inputs; composing two processes involves the extra complexity of specifying all the synchronization. The most classical way for real time

systems integration is the connection of classical programs using real-time Operating System (OS) primitives. Here, the main problem is the number of programs to analyse and connect: diagnostic and maintenance is difficult, temporal constraints are not expressed in the tasks (programs) description but are satisfied using the OS primitives for process synchronization/communication. This leads to systems which are generally non deterministic, and on which no safety properties can be formally guaranteed.

On the other side is the family of synchronous languages. The main drawbacks of a classical asynchronous implementation of reactive systems can be avoided using this class of languages. Resulting from the synchrony hypothesis, these languages are determistic, they allow concurrency and hierarchical specification. They provide safety, logical correctness (respect of the input/output specification), and temporal correctness. Furthermore they are based on a mathematically well defined semantics, supporting verification tools. For example, the compilation of SIGNAL code provides a graph on which static correctness proofs can be derived. It can also produce an equivalent dynamical equations system, on which dynamical properties can be proved. Using the tool SIGALI, the absence of deadlocks or properties specific to the application can be checked. In Reference [22], a variety of formal specifications and implementations of a real-time reactive system are proposed. This book contains a comparative survey of various languages (among which SIGNAL [15], and also LUSTRE, ESTEREL, STATECHARTS, ADA, ...) used to specify, verify and implement a controller for a robotic production cell.

An approach related to the integration of data-flow and sequencing of SIGNAL*GTi* is ARGOLUS [23]. It integrates ARGOS (hierarchical parallel automata) with LUSTRE (data flow); in SIGNAL*GTi* sequencing is specified in a more declarative style. A general study of preemption and concurrency, lead in combination with the imperative synchronous language ESTEREL, resulted in the possibility to control the starting, suspension, resuming and termination of external tasks [24]. However, the fact that these tasks can not be defined within the same language framework limits the control on interactions between different levels.

From the point of view of robot programming, the approach of applying synchronous languages is adopted in the ORCCAD environment [25]. This approach aims at a complete design environment for robot programming, and has more general goals than ours, which is focused on robot vision and the application of the SIGNAL environment as it is. However, task-level programming is done using ESTEREL [14], and the data-flow specification of control laws is done using another formalism at a different level, and controlled as an external task

as said before.

Discussion

SIGNAL is a data-flow language, and as such not adapted to all kinds of computation. For example, image processing or linear algebra cannot generally be performed with SIGNAL (or only with difficulty). Arrays are available in SIGNAL, but the algorithms involved for this kind of computation (for example the inverse of a matrix) are not naturally data-flow. External functions written with other languages (C and Fortran for instance), can be called from the SIGNAL program. The same method is used in our application for the management of the set of segments which is obviously not the application domain of synchronous languages. However, the use of such functions is not performed asynchronously: they are considered as any function defined in SIGNAL, thus the synchronous framework is not left. Furthermore, the management of asynchronous inputs or interruptions is not supported. However, this is not necessary in this kind of application where the inputs are provided regularly and periodically (here at video rate). Finally, dynamical management of time at the execution is not treated here, but this is not necessary due to the regular aspect of the loops.

Let us now emphasize the merits of synchronous languages, and more particularly SIGNAL, for this kind of applications. Dealing with implementation issues, advantages can be found at both control and task level. The data flow framework is particularly appropriate for the specification of visual servoing because of the equational and data flow nature of the closed-loop control laws, which can be implemented as control functions between sensor data and control outputs. The possibility of implicitly specifying parallel behaviors has been proved useful for the 3D structure estimation using active vision. The synchrony hypothesis corresponds well to the model of time in the equations defining the control laws. The second point concerns tasks sequencing and preempting. The language-level integration of the data flow and sequencing frameworks have been achieved as an extension of SIGNAL: SIGNAL *GTi*. It enables the definition of time intervals, their association with data flow processes and provides constructs for the specification of *hierarchical preemptive tasks*. This way, it offers a multi-paradigm language combining the data flow and multi-tasking paradigms, for hybrid applications blending (sampled) continuous and discrete transition aspects. SIGNAL *GTi* can be used for the design of a *hierarchy* of parallel automata, with the advantages of both the automata (determinism, tasks sequencing) and concurrent programming languages (parallelism between tasks) without their drawbacks. Note that SIGNAL *GTi* has not been

developed only for the application presented in this paper, but for the specification of other complex applications (such as behavioral animation in computer graphics [26] or the design of a transformer power station [27]).

The semantics of SIGNAL is also defined via a mathematical model of multiple clocked flows of data and events. SIGNAL programs describe relations on such objects: in that sense, programming is done via constraints. The compiler calculates the solutions of the system and may thus be used as a proof system. Its programming environment, which is not limited to the compiler, features tools for the automated analysis of formal properties. The compilation of SIGNAL code provides a dependencies graph on which static correctness proofs can be derived: it checks *automatically* the network of dependencies between data flows and detects causal cycles, temporal inconsistencies from the point of view of time indexes. SIGNAL synthesizes *automatically* the scheduling of the operations involved inside a control-loop (note that this work is an error-prone task when done by hand in classical C-like languages), and this scheduling is proved to be *correct* from the point of view of data dependencies. Furthermore, the SIGNAL-code is thus easy to modify since the re-synthesis is automatic. Finally, the compiler synthesizes *automatically* a global optimization of the dependencies graph.

The SIGNAL environment provides other tools (note that they have not been used directly in our application; see [15]): SIGNAL can produce an equivalent dynamical equation system, on which dynamical properties can be proved. The absence of deadlocks (liveness), reachability of states (or on the contrary non-reachability of a "bad" state), or properties specific to the application can thus be checked. These properties (both static and dynamic) checking tools are important at two levels: for development purposes it is important to verify that the system really has the expected or required behavior; and for the certification of the safety of the systems, which is meaningful regarding safety-critical applications. Research is going on concerning the distribution of SIGNAL programs on parallel machines, with automatic generation of separate code modules and of their communications. Finally, the compilation of SIGNAL into VHDL opens the ways towards hardware/software co-design. An environment with such tools provides effective assistance in the context of software engineering; other classical asynchronous languages do not offer them, while other synchronous languages like ESTEREL or LUSTRE do.

As a conclusion, the contribution of the synchronous approach, and of SIGNAL in particular, is that it has a programming style closer to a control engineer's specification and that

it provides him with a set of tools relieving him from error-prone tasks. Even if some other languages are sometimes provided with interesting other functionalities (management of the duration of tasks, dynamic scheduling, ...), they do not offer the ones mentioned here, based on the synchronous model.

CONCLUSION

The objective of this paper is to report on an experiment showing that synchronous languages are suitable for specifying and implementing vision tasks at different levels: camera motion control (vision-based closed loop control), perception task (structure estimation from controlled motion), and application (vision task sequencing). The first question addressed is to examine what the advantages are of using a data-flow synchronous language for programming visual servoing. The data flow paradigm is particularly adequate and suitable because of the equational and data flow nature of the closed loop control laws, which can be implemented as control functions between sensor data and control output. The possibility of specifying implicitely parallel behaviors proved useful when adding structure estimation. The synchrony hypothesis corresponds well to the model of time in the equations defining the control laws, and it is used by the compiler to perform a static verification of the logical timing correctness.

The second point concerns the specification of more complex applications, involving transitions between modes, *i.e.* the sequencing of data-flow tasks. The language-level integration of the data flow and task preemption paradigms is made in an extension to SIGNAL: SIGNAL *GTi*. It enables the designation of time intervals, their association with data flow processes in order to form tasks, and the sequencing of these data flow tasks. This way, the whole application can be specified in SIGNAL, from the discrete event driven transition behavior down to the (sampled) continuous servoing loop. The synchrony hypothesis corresponds well to the model of time in the equations defining the control laws, and it is used by the compiler to perform a static verification of the logical timing correctness. Implementation and experiments have been carried out on a robotic cell.

As a perspective in the direction of robot programming, it would be interesting to consider a generalization of the structure of data-flow tasks proposed in this paper, towards a programming environment dedicated to the design of sensor-based control tasks following the task-function approach, as presented in the perspectives of [25].

Acknowledgment

This work was partly supported by the CNRS inter-PRC project VIA (*Vision Intentionnelle et Action*) and by the MESR under contribution to a student grant. The authors wish to thank Samuel Ketels and Florent Martinez who implemented part of the experimental work.

References

[1] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proc. of the IEEE*, 79(9):1270–1282, September 1991.

[2] D. Harel, A. Pnueli. On the development of Reactive Systems In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, vol. 13 of NATO ASI Series, pp. 477-498, Springer Verlag, 1985.

[3] P.J. Ramadge, W.M. Wonham. The Control of Discrete Events Systems. *Proc. of the IEEE*, 77(1):81–97, January 1989.

[4] T.P. Baker, O. Pazy. Real time features for ADA9X. In *Proc. of the IEEE Real Time Systems Symposium*, pp. 172–180, December 1991.

[5] G. Berry. Real-time programming: special purpose languages or general purpose languages. In *Proc. of the 11^{th} IFIP World Congress*, pp. 11–17, San Fransisco, California, 1989.

[6] F. Boussinot and R. de Simone. – The ESTEREL language. – *Proc. of the IEEE*, 9(79):1293–1304, September 1991.

[7] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer, 1993.

[8] P. Le Guernic, M. Le Borgne, T. Gautier, and C. Le Maire. Programming real time application with SIGNAL. *Proc. of the IEEE*, 79(9):1321–1336, September 1991.

[9] D. Harel. STATECHARTS: a visual formalism for complex systems. *Science of Computer programming*, 8(3):231–274, 1987.

[10] A. Benveniste. Synchronous languages provide safety in reactive systems design. *Control Engineering*, pp. 87–89, September 1994.

[11] E. Rutten and P. Le Guernic. The sequencing of data flow tasks in SIGNAL. In *Proc. of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, Orlando, Florida, June 1994. (ftp `cs.umd.edu`, file `/pub/faculty/pugh/sigplan_realtime_workshop_94/rutten.ps.Z`).

[12] Y. Sorel. Massively parallel computing systems with real-time constraints: the "algorithm-architecture adequation" methodology. In *Proc. of the Massively Parallel Computing Systems Conference*, Ischia, Italy, 1994.

[13] E. Marchand, E. Rutten, and F. Chaumette. *From Data-Flow Task to Multi Tasking: Applying the Synchronous Approach to Active Vision in Robotics.* Accepted for publication in *IEEE Trans. on Control Systems Technology*, 1996.

[14] E. Coste-Manière, B. Espiau, and E. Rutten. A task-level robot programming language and its reactive execution. In *IEEE Int. Conf. on Robotics and Automation*, pp. 2751–2737, Nice, France, May 1992.

[15] T. Amagbegnon, P. Le Guernic, H. Marchand, and E. Rutten. SIGNAL – the specification of a generic, verified production cell controller. In [22]. (chap. VII, pp. 115 – 129)

[16] F. Chaumette, S. Boukir, P. Bouthemy, and D. Juvin. Structure from controlled motion. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 18(5):492–504, May 1996.

[17] B. Espiau, F. Chaumette, and P. Rives. A new approach to visual servoing in robotics. *IEEE Trans. on Robotics and Automation*, 8(3):313–326, June 1992.

[18] E. Marchand and F. Chaumette. Controlled Camera Motions for Scene Reconstruction and Exploration. In *IEEE Int. Conf. on Computer Vision and Pattern Recognition, CVPR'96*, pp. 169–176, San Francisco, CA, USA, June 1996.

[19] A.D.H. Thomas, M.G. Rodd, J.D. Holt, C.J. Neill. Real Time Industrial Visual Inspection: A Review. *Real Time Imaging*, 1:139-15, 1995.

[20] C.AS.R. Hoare. *Communicating Sequential Systems.* Prentice-Hall, 1985.

[21] R. Milner. *A calculus of communicating systems.* LNCS n^{o} 92, Springer, 1980.

[22] C. Lewerentz, T. Lindner, editors. *Case Study Production Cell – A Comparative Study in Formal Software Development*, LNCS no. 891, Springer, January, 1995.

[23] M. Jourdan, F. Lagnier, F. Maraninchi, and F. Raymond. A multiparadigm language for reactive systems. In *Proc. of the IEEE Int. Conf. on Computer Languages, ICCL'94*, Toulouse, France, May 1994.

[24] G. Berry. Preemption in concurrent systems. In *Proc. of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, Bombay, India, December 1993. LNCS n^{o} 761, Springer.

[25] D. Simon, B. Espiau, E. Castillo, and K. Kapellos. Computer-aided design of a generic robot controller handling reactivity and real-time controller issues. *IEEE Trans. on Control Systems Technology*, 1(4):213–229, December 1993.

[26] S. Donikian, E. Rutten. Reactivity, concurrency, data-flow and hierarchical preemption for behavioral animation. In R.C. Veltkamp, E.H. Blake (eds.), *Programming Paradigms in Graphics*, Springer, Computer Science, 1995. (pp. 137–153,169)

[27] H. Marchand, E. Rutten, M. Samaan. Synchronous design of a transformer station controller with SIGNAL. In *Proceedings of the 4th IEEE Conference on Control Applications, CCA'95*, Albany, New York, September 28–29, 1995. (pp. 754–759)